INVESTIGATING METRIC SPIKE

TASK - 3

BY NAZIREEN SANIA

CASE STUDY : INVESTIGATING METRIC SPIKE

For the first section of the project, we work with 3 tables:

- **users**: Contains one row per user, with descriptive information about that user's account.
- **events**: Contains one row per event, where an event is an action that a user has taken (e.g., login, messaging, search).
- **email_events**: Contains events specific to the sending of emails.

METHODOLOGY

We first load the tables into MySQL. First we create a database named 'project3' and create the tables 'users', ' emails' and 'email_events' under it.

We then determine the file paths to store our excel file using the 'secure_file_priv' function. Before we load the files, we check for any missing values using the 'Select & Find' option. We do this instead of Table Data Import Wizard option because the values were not getting loaded completely on using that option.After determining the file paths, we load the data using the 'load data infile' command.

```
#creating table1#
create table `users` (
`user_id` INT ,
`created_at` VARCHAR(100),
`company_id` INT,
`language` VARCHAR(50),
`activated_at` VARCHAR(100),
`state` VARCHAR(50)
);
```

```
load data infile 'C:\\ProgramData\\MySQL\\MySQL Server 8.0\\Uploads\\users.csv'
into table `users`
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS
(`user_id`,`created_at`,`company_id`,`language`,`activated_at`,`state`);

select count(*) from users;

SET SQL_SAFE_UPDATES = 0;

#updating the cretaed_at column from varchar to date datatype#
alter table users add column temp_created_at  datetime;
update users set temp_created_at = STR_TO_DATE(created_at, '%d-%m-%Y %H:%i');
alter table users drop column created_at;
alter table users change column temp_created_at created_at datetime;


#updating the activated_at column from varchar to date datatype#
alter table users add column temp_activated_at  datetime;
update users set temp_activated_at = STR_TO_DATE(activated_at, '%d-%m-%Y %H:%i');
alter table users drop column activated_at;
alter table users change column temp_activated_at activated_at datetime;

select * from users;
```

We also convert the columns containing date from varchar datatype to date datattype using STR_TO_DATE function in MySQL. We repeat the process for all the tables. Once all entries are loaded, we begin the tasks.

DATA ANALYSIS TASKS

(1) Weekly User Engagement

Objective: Measure the activeness of users on a weekly basis.

```sql
#(1) Measure the activeness of users on a weekly basis#

SELECT
    YEARWEEK(occurred_at, 1) AS `week`,
    user_id,
    COUNT(*) AS weekly_events
FROM events
GROUP BY user_id, `week`
ORDER BY `user_id`;
```

Result Grid | Filter Rows:

week	user_id	weekly_events
201420	4	4
201421	4	8
201422	4	29
201423	4	4
201424	4	15
201425	4	8
201426	4	7
201427	4	10
201428	4	8
201418	8	2
201419	8	15
201420	8	3
201421	8	9
201431	8	7
201425	11	34
201426	11	30
201431	11	53
201432	11	9
201431	17	38
201433	17	17

- YEARWEEK(occurred_at, 1) AS week
  1. Extracts the year and week number from occurred_at.
  2. YEARWEEK(date, 1) ensures ISO week format (Monday as the start of the week).
  3. Example: If occurred_at = '2024-03-22', this function will return 202412 (Year 2024, Week 12).
- user_id - Retrieves the ID of the user who performed the event.
- GROUP BY `week`, user_id :

Groups the data by week and user_id, meaning:

1. Each row in the result represents one user's activity in one specific week.
2. COUNT(*) will give us the total number of events per user per week.

- ORDER BY `user_id`;

- Sorts the results chronologically by user_id.
- Helps visualize trends over time.

(2) User Growth Analysis

Objective: Calculate user growth over time.

```sql
#(2) Calculate user growth over time #

SELECT
    YEARWEEK(created_at, 1) AS week,
    COUNT(user_id) AS new_users
FROM users
GROUP BY week
ORDER BY week;
```

- YEARWEEK(signup_date, 1) AS week
  1. Extracts the year and week number from signup_date.
  2. YEARWEEK(date, 1) ensures ISO week format (Monday as the start of the week).
  3. Example: If occurred_at = '2024-03-22', this function will return 202412 (Year 2024, Week 12).
- COUNT(user_id) AS new_users

1. Counts the number of new users who signed up in each week.

2. Since user_id is unique for each user, COUNT(user_id) gives us the number of new users.

- GROUP BY `week` :

Each row in the result represents one user's activity in one specific week.

- ORDER BY `week`;

- Sorts the results in ascending order of week number, so the earliest signups appear first.
- Helps visualize trends over time.

	A	B
1	week	new_users
2	201301	26
3	201302	29
4	201303	47
5	201304	36
6	201305	30
7	201306	48
8	201307	41
9	201308	39
10	201309	33
11	201310	43
12	201311	33
13	201312	32
14	201313	33
15	201314	40
16	201315	35
17	201316	42
18	201317	48
19	201318	48
20	201319	45
21	201320	55
22	201321	41
23	201322	49
24	201323	51
25	201324	51
26	201325	46
27	201326	57
28	201327	57
29	201328	52
30	201329	71
31	201330	66
32	201331	69

(till 87 entries)

(3) Weekly Retention Analysis

Objective: Calculate weekly retention based on signup cohorts.

```sql
#(3) Calculate weekly retention based on signup cohorts.

SELECT
    YEARWEEK(u.created_at, 1) AS signup_week,
    COUNT(DISTINCT e.user_id) AS retained_users
FROM users u
JOIN events e ON u.user_id = e.user_id
GROUP BY signup_week
```

- YEARWEEK(u.created_at, 1) extracts the signup week for each user.
- (e.user_id) counts distinct users who had any event meaning after signing up, meaning they were retained.
- JOIN events e ON u.user_id = e.user_id joins users and events tables on user_id to track engagement.
- The code analyzes retention trends by showing how many users remained engaged per signup week.

	A	B
1	signup_week	retained_users
2	201301	9
3	201302	9
4	201303	13
5	201304	18
6	201305	15
7	201306	21
8	201307	16
9	201308	19
10	201309	10
11	201310	20
12	201311	17
13	201312	14
14	201313	13
15	201314	16
16	201315	13
17	201316	17
18	201317	25
19	201318	28
20	201319	21
21	201320	24
22	201321	13
23	201322	12
24	201323	20
25	201324	22
26	201325	19

(till 87 rows)

(4)Weekly Engagement Per Device

Objective: Measure activeness of users on a weekly basis per device.

```
# Measure activeness of users on a weekly basis per device.#

SELECT
    YEARWEEK(occurred_at, 1) AS `week`,
    device,
    COUNT(DISTINCT user_id) AS active_users
FROM events
GROUP BY `week`, device
ORDER BY `week`;
```

- YEARWEEK(occurred_at, l) extracts the occurred week for each user.
- YEARWEEK(occurred_at,l) AS 'week' , 'device'groups user activity by week and device.
- COUNT (DISTINCT user_id) counts distinct active users per week and device
- ORDER BY week sorts results by week to analyze changes in user activity.
- The code tracks user engagement trends across different devices over time.

	A	B	C
1	week	device	active_users
2	201418	acer aspire desktop	10
3	201418	acer aspire notebook	21
4	201418	amazon fire phone	4
5	201418	asus chromebook	23
6	201418	dell inspiron desktop	21
7	201418	dell inspiron notebook	49
8	201418	hp pavilion desktop	15
9	201418	htc one	16
10	201418	ipad air	30
11	201418	ipad mini	21
12	201418	iphone 4s	21
13	201418	iphone 5	70
14	201418	iphone 5s	45
15	201418	kindle fire	6
16	201418	lenovo thinkpad	90
17	201418	mac mini	8
18	201418	macbook air	57
19	201418	macbook pro	154
20	201418	nexus 10	16
21	201418	nexus 5	43
22	201418	nexus 7	20
23	201418	nokia lumia 635	19
24	201418	samsumg galaxy tablet	8
25	201418	samsung galaxy note	7
26	201418	samsung galaxy s4	56
27	201418	windows surface	10
28	201419	acer aspire desktop	26
29	201419	acer aspire notebook	34
30	201419	amazon fire phone	9

(till 468 rows)

(5) Email Engagement Analysis

Objective: Calculate email engagement metrics.

```sql
#Calculate email engagement metrics.#

SELECT
    YEARWEEK(occurred_at, 1) AS week,
    COUNT(*) AS emails_sent,
    COUNT(CASE WHEN `action` = 'email_clickthrough' THEN 1 ELSE NULL END) AS email_clickthrough,
    COUNT(CASE WHEN `action` = 'email_open' THEN 1 ELSE NULL END) AS email_open,
    COUNT(CASE WHEN `action` = 'sent_reengagement_email' THEN 1 ELSE NULL END) AS sent_reengagement_email,
    COUNT(CASE WHEN `action` = 'sent_weekly_digest' THEN 1 ELSE NULL END) AS sent_weekly_digest
FROM email_events
GROUP BY week
ORDER BY week;
```

	A	B	C	D	E	F
1	week	emails_sent	email_clickthrough	email_open	sent_reengagement_email	sent_weekly_digest
2	201418	1525	187	332	98	908
3	201419	4119	434	919	164	2602
4	201420	4290	479	971	175	2665
5	201421	4405	498	995	179	2733
6	201422	4480	453	1026	179	2822
7	201423	4595	492	993	199	2911
8	201424	4796	533	1070	190	3003
9	201425	5063	563	1161	234	3105
10	201426	5008	524	1090	187	3207
11	201427	5251	559	1168	222	3302
12	201428	5465	622	1230	214	3399
13	201429	5592	607	1260	226	3499
14	201430	5593	584	1211	206	3592
15	201431	5955	633	1386	230	3706
16	201432	5767	432	1336	206	3793
17	201433	5908	430	1357	224	3897
18	201434	6177	487	1421	257	4012
19	201435	6400	493	1533	263	4111

- YEARWEEK(occurred_at, l) extracts the occurred week for each user.
- YEARWEEK(occurred_at,l) AS 'week' , 'device'groups user activity by week.
- COUNT(*) AS emails_sent ccounts total emails sent which includes all email-related events.
- The COUNT CASES(s) categorizes emails based on actions (email_clickthrough, email_open, etc.) using COUNT(CASE WHEN action = ... THEN l ELSE NULL END).
- ORDER BY week sorts results by week to analyze email engagement trends over time.

ADDITIONAL TASKS

(1) Subqueries

Objective: Get users who performed more events than the average number of events.

```sql
# Get users who performed more events than the average number of events.

SELECT user_id, COUNT(*) AS event_count
FROM events
GROUP BY user_id
HAVING COUNT(*) > (
    SELECT AVG(event_count)
    FROM (
        SELECT COUNT(*) AS event_count
        FROM events
        GROUP BY user_id
    ) AS subquery
);
```

user_id	event_count
10522	58
10612	250
10736	56
11020	78
11227	278
11231	111
11261	344
11284	363
11301	83
11376	59
11395	80
11536	86
11562	63
11592	129
11599	64
11602	137

- COUNT(*): Counts total events per user.
- GROUP BY user_id: Groups rows to aggregate by each user.
- Subquery in HAVING: Computes the average event count across users.
- HAVING: Filters after grouping – used here to compare user event count with the average.

(2) Creating a View

Objective: Create a view to monitor weekly active users per device

```
# Create a view to monitor weekly active users per device.


CREATE VIEW weekly_device_activity AS
SELECT
    YEARWEEK(occurred_at, 1) AS `week`,
    device,
    COUNT(DISTINCT user_id) AS active_users
FROM events
GROUP BY `week`, device;
```

week	device	active_users
201418	acer aspire desktop	10
201418	acer aspire notebook	21
201418	amazon fire phone	4
201418	asus chromebook	23
201418	dell inspiron desktop	21
201418	dell inspiron notebook	49
201418	hp pavilion desktop	15
201418	htc one	16
201418	ipad air	30
201418	ipad mini	21
201418	iphone 4s	21
201418	iphone 5	70
201418	iphone 5s	45
201418	kindle fire	6
201418	lenovo thinkpad	90
201418	mac mini	8
201418	macbook air	57
201418	macbook pro	154
201418	nexus 10	16
201418	nexus 5	43

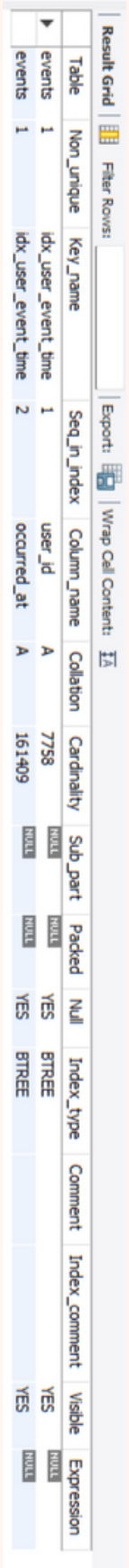
- CREATE VIEW: Defines a virtual table you can query like a real table.
- YEARWEEK(occurred_at, 1): Extracts week and year from the timestamp.
- COUNT(DISTINCT user_id): Counts unique users per week and device.
- GROUP BY: Groups results by both week and device.

(3) Creating an Index

Objective: Optimize queries filtering by user_id and occurred_at.

```
# Optimize queries filtering by user_id and occurred_at.


CREATE INDEX idx_user_event_time ON events(user_id, occurred_at);
SHOW INDEX FROM events;
```



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
events	1	idx_user_event_time	1	user_id	A	7758	NULL	NULL	YES	BTREE			YES	NULL
events	1	idx_user_event_time	2	occurred_at	A	161409	NULL	NULL	YES	BTREE			YES	NULL

- CREATE INDEX: Improves query speed for searches, filters, and joins.
- user_id, occurred_at: Index applied to these columns – useful when queries filter or join using them.
- The index gets used automatically by MySQL during query execution if beneficial.